REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-00-

0405

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REP... 15 December 1996-14 December 1999

4. TITLE AND SUBTITLE
Mobile Agents and Systems Principles

5. FUNDING NUMBERS
F49620-97-1-0013

6. AUTHOR(S)
Fred B. Schneider
Gregory Morrisett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Department
Cornell University
Ithaca, New York 14853

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR
801 North Randolph Street, Room 732
Arlington, VA 22203-1977

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-97-1-0013

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Under the auspices of this AFOSR funding, research was performed on a variety of topics related to the implementation of fault-tolerant and secure systems, with emphasis on systems that are extensible and/or that employ mobile code. The research involved theoretical as well as practical components and led to 32 publications (listed at the end of this report), including two books, and two patents.

14. SUBJECT TERMS			15. NUMBER OF PAGES 9
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

FA 3-23-00

Mobile Agents and Systems Principles

AFOSR Grant F49620-97-1-0013

Final Technical Report

1 August 96 – 14 December 99

Fred B. Schneider
Computer Science Department
Cornell University
Ithaca, New York
(607) 255-9221 (phone)
(607) 255-4428 (fax)
fbs@cs.cornell.edu

Gregory Morrisett
Computer Science Department
Cornell University
Ithaca, New York
(607) 255-3009 (phone)
(607) 255-4428 (fax)
jgm@cs.cornell.edu

Research Accomplishments

Under the auspices of this AFOSR funding, research was performed on a variety of topics related to the implementation of fault-tolerant and secure systems, with emphasis on systems that are extensible and/or that employ mobile code. The research involved theoretical as well as practical components and led to 32 publications (listed at the end of this report), including two books, and two patents.

Agent Integrity

Agents comprising an application must not only survive (possibly malicious) failures of the hosts they visit, but they must also be resilient to hostile actions by other hosts. Replication and voting enable an application to survive some failures of the hosts it visits. Hosts that are not visited by agents of the application, however, can masquerade and confound a replication scheme. Two classes of protocols to solve these *agent integrity* problems were initially developed as part of this AFOSR project. One class uses chained cryptographic certificates; the second class uses cryptographic signature-sharing.

We were then able to unify these protocols by viewing them in terms of delegation. In each, the principals are sets of hosts (services) and authorization is transferred from one principal to another.

In some settings, hosts being visited by agents cannot be replicated, so the preceding protocols do not apply. This led us to investigate protocols for agent fault-tolerance without host replication.[1] With these *NAP protocols*, execution of an agent A on a host is monitored by agents (napping) on other hosts. If the failure of A or of the host on which A executes is detected, then one of the napping agents performs a recovery action. This recovery action might involve retrying A, dispatching a different agent to some other host, or alerting the computation's initiator of a problem. NAP is not resilient to hostile host failures, but without using replication no scheme can be.

The difficult part of implementing NAP involves coordinating the napping agents. A protocol that tolerates multiple failures must have multiple agents napping, each monitoring execution. A coordination protocol is required to ensure that more than one napping agents does not detect and try to restart a failed agent. Our initial solutions to the coordination problem were complex enough that their correctness was suspect. This led us to show that the problem was actually an instance of the (fail-stop) reliable broadcast problem that we solved in 1983. And, by refining our 1983 protocol, we were able to support a broad class of strategies for how napping agents are disbursed in the network. This broader class of strategies allows our protocols also to work when the trajectory of an agent folds back on itself, visiting a host that is still running a napping agent.

Enforceable Security Policies

A *security policy* defines executions that, for one reason or another, have been deemed unacceptable. To date, application-independent security policies—like mandatory and discretionary access control, information flow restrictions, and resource availability—have attracted most of the attention. But with the expanding role of computers in our infrastructure, specialized, application-dependent security policies are becoming increasingly important. For example, a system to support mobile code might prevent information leakage by enforcing a security policy that bars messages from being sent after files are read. To support electronic commerce, a security policy might prohibit executions in which a customer pays for a service but the seller does not provide that service.

[1] This work is joint with Dag Johansen at the University of Tromsoe (Norway) and Keith Marzullo at the Univ of California, San Diego.

Over the period of this grant, we developed a mathematical characterization of what security policies are enforceable. First, we proved that enforcement mechanisms cannot exist for security policies that are not safety properties. Second, we developed a new class of enforcement mechanisms and proved that it is complete for the set of all enforceable security policies, which turns out to be the set of safety properties. Our new class of mechanisms is based on *security automata*, automata that accept finite and infinite sequences.

A security automaton serves as an enforcement mechanism for some target system by monitoring and controlling the execution of that system. Each action or new state corresponding to a next step that the target system takes is sent to the security automaton and serves as the next symbol of that automaton's input. If the automaton cannot make a transition on an input symbol, then the target system is about to violate the security policy specified by the automation, and the target system is terminated.

We demonstrated the practicality of enforcing security policies expressed using security automata by constructing and evaluating tools to generate *inlined reference monitors* that implement security automata for both the Java Virtual Machine and Intel x86 machines. The first prototype (SASI) worked for programs written or compiled into Java virtual machine code (JVML) or Intel's x86 machine code; a second generation (PoET/PSLang) refined the approach for JVML. Specifically, given a security automaton SA that expresses a security policy and given a machine language program P, both SASI and PoET/PSLang add checks to P that are necessary in order to ensure that executing P is guaranteed not to violate the security policy defined by SA. In addition, using standard compiler analyses, our prototypes attempt to minimize the number of checks inserted.

Using SASI, we experimented with generalizations of two well known security policies: software fault isolation (SFI) and the Java Standard Security Manager. Our experiments confirmed that SASI generates code comparable with hand-coded, heavily optimized SFI tools for the x86, and in fact exceeds the performance of the hand-coded Java Standard Security Manager. Furthermore, security automaton specifications of the security policies have proven to be easy to write, understand, and modify. Using PoET/PSLang, we showed how to support the Java 2 "stack inspection" security policy without any support from the Java virtual machine. This, for example, allows Java 2 programs to be executed on previous generations of the Java run-time system; it also allows deployment of variations and refinements of the Java security policy.

Inter-agent and Host Security

Not only must agents be protected from attack by hosts, but hosts and agents must be protected from attack by other agents. Traditional approaches to ensuring host and inter-agent security employ a reference monitor or interpreter to execute operations on behalf of untrusted agents. However, the run-time costs of this can be prohibitive. We therefore investigated an alternative approach based on *proof carrying code* (PCC). In the general PCC framework, each agent comprises raw machine code and a formal proof that the code will not violate the security policy of interest. Before executing an agent, a host checks the proof. To make the system practical, we concentrated on *type safety* as the security policy and constructed a compiler that automatically produces native code and a proof of type safe from a high-level, strongly-typed language. When the type system is sufficiently powerful to encode the security policy, no run-time cost is incurred; and when the type system is too weak, run-time checks are inserted in the code, but the type system ensures that the checks cannot be bypassed.

The first step was to design a strongly typed assembly language (TAL) suitable for use as a target language for programming agents. The type system for this assembly language is powerful enough to ensure that a wide variety of safety properties are satisfied. For example, if a host provides an interface for the abstract type *file descriptors* along with operations for creating and manipulating file descriptors, then type correct program agents cannot forge file descriptors nor can they apply any operations to file descriptors except those exported by the host.

Many high-level languages, such as Java and SML, provide similar guarantees. However, TAL has two important properties that make it more appealing in environments with mobile code. First, as an assembly language, TAL can serve as a target language for a variety of high-level language compilers, including those for both Java and SML. Agents that are compiled to TAL need not be tied to a particular language environment. Second, all compilation, including low-level optimizations like copy propagation, register allocation, and instruction scheduling, can be performed before shipping agents to a host. Thus, unlike current Java implementations, a host need not have access to a trusted high-level language compiler or interpreter in order to support an agent. Furthermore, there is no need to pay the overhead of interpretation or compilation before invoking an agent.

To demonstrate the expressiveness of TAL, we showed how to compile a high-level ML-based language to TAL, and we have proven that such a compiler "preserves typing". Specifically, we established:

- The compiler always produces well-typed assembly language.
- Type abstractions at the source level are preserved at the assembly language level.

Thus, for example, a host might use an ML abstract type to implement file descriptors, compile the ML code to TAL, and the type system of TAL will ensure that no agent can forge descriptors as described above.

The design principles behind our idealized typed assembly language were used to provide a concrete implementation of these ideas. In particular, we constructed a suite of tools for automatically type-checking annotated Intel x86 assembly language (and object files). We also constructed tools to verify that object files match specified interfaces and, therefore, may be safely linked to form a well-typed executable program.

To evaluate the claims that a typed assembly language provides both language independence and support for highly-optimized code, we constructed a set of prototype compilers that map high-level language programs to type-correct assembly code. Specifically, we constructed prototype Scheme and Safe-C compilers that generate type-correct Intel x86 code. In addition, we built a number of language-independent optimizations such as graph-coloring register allocation, in order to identify shortcomings in the type system that prevent optimization and to compare our approach quantitatively to others.

Progress was also made in the design of the type system that could express additional security properties as typing assertions. For instance, we studied typing mechanisms to support a purely static form of *capability enforcement.* Like traditional capability-based systems, our type system allows agents to access objects only if they have a capability to do so. Capabilities may be granted by the host and are unforgeable by agents. However, unlike traditional capability systems, agents need not carry and present capabilities at run-time. Rather, as with types, capabilities are a purely static concept and hence incur no dynamic overhead.

The drawback of this approach is that capabilities cannot always be revoked by the host. Instead, the host and the agent must agree to revoke a capability. Nevertheless, we found compelling applications for such a mechanism, including static verification of memory management (garbage collection) and concurrency control.

Publications

(1) Morrisett, G. and F. Smith. Mostly Copying and Conservative Mark-Sweep Collection. *International Symposium on Memory Management*, Vancouver, CA, Oct. 1998, 68-78.

(2) Schneider, F.B. On Traditions in Marktoberdorf. *Deductive Program Design* (M. Broy, ed.) ASI Vol. F152. Springer-Verlag, Heidelberg, 1–4.

(3) Schneider, F.B. Notes on Proof Outline Logic. *Deductive Program Design* (M. Broy, ed.) ASI Vol. F152. Springer-Verlag, Heidelberg, 351–394.

(4) Schneider, F.B. *On Concurrent Programming.* Springer Verlag, NY, 1997, 473 pages.

(5) Schneider, F.B. (ed.) Information Systems Trustworthiness — Interim Report. Computer Science and Telecommunications Board Commission on Physical Sciences, Mathematics, and Applications National Research Council. April 1997.

(6) Dolev, D., R. Reischuk, F.B. Schneider, and H.R. Strong. Report on Dagstuhl Seminar on Time Services, Schloss Dagstuhl, March 11–March 15 1996. *Real-Time Systems 12*, 3 (May 1997), 329–345.

(7) Schneider, F.B. Editorial: New Partnership with ACM. *Distributed Computing 10*, 2 (1997), 63.

(8) Minsky, Y., R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. *Proc. of the Seventh ACM SIGOPS European Workshop "System Support for Worldwide Applications"* (Connemara, Ireland, Sept. 1996), ACM, New York, 109–114.

(9) Johansen, D., R. van Renesse, and F.B. Schneider. Supporting Broad Internet Access to TACOMA. *Proc. of the Seventh ACM SIGOPS European Workshop "System Support for Worldwide Applications"* (Connemara, Ireland, Sept. 1996), ACM, New York, 55–58.

(10) Stoller, S.D. and F.B. Schneider. Automated Analysis of Fault-Tolerance in Distributed Systems. *Proc. of the First ACM SIGPLAN Workshop on Automated Analysis of Software* (R. Cleaveland and D. Jackson, eds.) (Paris, France, Jan. 1997) ACM, New York, 33–44.

(11) Stoller, S.D. and F.B. Schneider. Automated Stream-Based Analysis of Fault-Tolerance. *Formal Techniques in Real-time and Fault-Tolerant Systems (FTRTFT '98)*, (September 1998, Lyngby, Denmark), Lecture Notes in Computer Science, Volume 1486, Springer Verlag, Berlin, 1998, 113–122.

(12) Morrisett, G., D. Walker, and K. Crary. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3), May 1999, pp. 528–569. An earlier version appeared in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, TX, Jan. 1998), ACM 85–97.

(13) Crary, K., D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, TX, Jan. 1999), ACM, 262-275.

(14) Glew, N. and G. Morrisett. Type-Safe Linking and Modular Assembly Language. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 1999), ACM, 250-261.

(15) Johansen, D., R. van Renesse, and F.B. Schneider. Operating System Support for Mobile Agents. Republished in *Readings in Agents* (M.N. Huhns and M.P. Singh eds.), Morgan Kaufman Publishers, San Francisco, CA, 1997, 263–266.

(16) Morrisett, G. and R. Harper. Typed Closure Conversion for Recursively-Defined Functions (Extended Abstract). In *Higher-Order Operational Techniques in Semantics (HOOTS) II* (A. Gordon, A. Pitts, and C. Talcott, ed.) Vol. 10, Electronic Notes in Theoretical Computer Science, Elsevier, 1998.

(17) Schneider, F.B. (editor). *Trust in Cyberspace.* National Academy Press, Washington, D.C. 1999, 331 pages.

(18) Schneider, F.B. On Concurrent Programming. Invited "Inside Risks" column. *Communications of the ACM 41*, No. 4 (April 1998), 128.

(19) Gries, D. and F.B. Schneider. Adding the everywhere operator to propositional logic. *Journal of Logic and Computation 8*, No. 1 (Feb. 1998), 119–129.

(20) Schneider, F.B. Towards Fault-Tolerant and Secure Agentry. Invited paper. *Proceedings of the 11th International Workshop WDAG '97*, Saarbrucken, Germany, Sept. 1997. Lecture Notes in Computer Science, Volume 1320, Springer-Verlag, Heidelberg, 1997, 1–14.

(21) Morrisett, G., K. Crary, N. Glew, and D. Walker. Stack-Based Typed Assembly Language. *Proceedings of the ACM Workshop on Types in Compilation* (Kyoto, Japan, Mar. 1998), Xavier Leroy and Atsushi Ohori, editors, Lecture Notes in Computer Science, Volume 1473, Springer-Verlag, Heidelberg, 1998, 28-52.

(22) Crary, K., S. Weirich, and G. Morrisett. Intentional Type Analysis in Type Erasure Semantics. *ACM SIGPLAN International Conference on Functional Programming* (Baltimore, MD, Sep. 1998), ACM, 301-312.

(23) Schneider, F.B. Improving Networked Information System Trustworthiness: A Research Agenda. *Proceedings 21st National Information Systems Security Conference* (October 1998, Arlington, Virginia), National Computer Security Center, 766.

(24) Smith, F. and G. Morrisett. Comparing Mostly-Copying and Mark-Sweep Conservative Collection. *Proceedings of the 1998 ACM SIGPLAN International Symposium on Memory Management*, Vancouver, BC, Oct. 1998.

(25) Schneider, F.B. Towards Trustworthy Networked Information Systems. Invited "Inside Risks" column. *Communications of the ACM 41*, 11 (November 1998), 144.

(26) Schneider, F.B. and S.M. Bellovin. Evolving Telephone Networks. Invited "Inside Risks" column. *Communications of the ACM 42*, No. 1 (Jan. 1999), 160.

(27) Schneider, F.B., D. Johansen, R. van Renesse. What Tacoma Taught Us. *Mobility: Processes, Computers, and Agents*, Dejan S. Milojicic, Frederick Douglis, and Richard G. Wheeler (eds.), Addison Wesley and the ACM Press, April 1999, 564–566.

(28) Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A Realistic Typed Assembly Language. *ACM SIGPLAN Workshop on Compiler Support for System Software* (Atlanta, GA, May 1999), ACM, 25-35.

(29) Johansen, D., K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical Fault-tolerance for Itinerant Computations. *Proc. 19th IEEE International Conference on Distributed Computing Systems* (June 1999, Austin, Texas), IEEE, 180–189.

(30) Zdancewic, S., D. Grossman, and G. Morrisett. Principals in Programming Languages: A Syntactic Proof Technique. *1999 ACM SIGPLAN International Conference on Functional Programming*, (Paris, France, Sep. 1999), ACM, 197-207.

(31) Erlingsson, U. and F.B. Schneider. SASI enforcement of security polices: A retrospective. *Proceedings New Security Paradigms Workshop* (Ontario, Canada, Sept 1999), ACM.

(32) Schneider, F.B., S. Bellovin and A. Inouye. Building trustworthy systems: Lessons from the PTN and Internet. *IEEE Internet Computing*, 3, 5 (November-December 1999), 64-72.

Patents

(1) Transparent fault tolerant computer system. United States Patent 5,802,265, Sept. 1, 1998. Co-inventors: T.C. Bressoud, J.E. Ahern, K.P. Birman, R.C.B. Cooper, B.Glade, and J.D. Service.

(2) Transparent fault tolerant computer system. United States Patent 5,968,185, Oct. 19, 1999. Co-inventors: T. C. Bressoud, J. E. Ahern, K. P. Birman, R. C. B. Cooper, B. Glade, and J. D. Service.